Constructing Specifications by Combining Parallel Elaborations

MARTIN S. FEATHER

Abstract—**Constructing specifications of complex tasks is often a laborious activity in spite of the rich vocabulary provided by specification languages. An incremental approach to construction is proposed, with the virtue of offering considerable opportunity for mechanized support. Following this approach one builds a specification through a series of elaborations that incrementally adjust a simple initial specification. Elaborations perform both refinements, adding further detail, and adaptations, retracting oversimplifications and tailoring approximations to the specifics of the task. It is anticipated that the vast majority of elaborations can be concisely described to a mechanism which will then perform them automatically. When elaborations are independent, they can be applied in parallel, leading to diverging specifications which must later be recombined.**

The approach is intended to facilitate comprehension and maintenance of specifications, as well as their initial construction. The advantages of following this approach stem from the *gradual* nature of the elaboration process, the *separation of concerns* through following independent elaborations in parallel, the *simplicity* of the individual elaboration steps (the effects of each step are well delineated), and the availability of an *explicit record of construction*.

Index Terms—**Elaboration of designs, explanation, high-level editing, maintenance, reuse, specification evolution.**

I. Introduction

A. The Role of Formal Specifications in Software Development

A NUMBER of researchers have argued forcefully that the best way to significantly improve the production and maintenance of software is to formalize and provide mechanical assistance for the entire programming process (see the joint report [13]). The acquisition and use of a *formal specification* of the task to be programmed is a common factor to many of the methods proposed to attain this goal.

Such specifications serve as formal expressions of informal requirements. A formal specification differs from the intended program by expressing only *what* the desired behavior of that program is to be, without being constrained to state *how* the program is to achieve that behavior. Because of the specification's formal nature, mechanical tools can assist in its use in a number of ways:

Contract: The specification serves as a reference point against which to check a candidate implementation.

Manuscript received December 30, 1986; revised June 30, 1988. This work was supported in part by Rome Air Development Center under Contract F30602-85-C-0221 and in part by the National Science Foundation under Contract MCS-7918792.

The author is with USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292.

IEEE Log Number 8825085.

Analysis: Studies of the specification (via checking for consistency and completeness, theorem proving, and symbolic evaluation) provide feedback on some of the implications of the task that are not readily apparent at the level of informal requirements.

Prototyping: The specification is executed to reveal some aspects of the operation of the proposed software. Such execution generally falls short of the implementation in some way, for example, by using resources and facilities which will not be available to the actual implementation, or by crudely approximating the user interface and so forth.

Derivation: The formal specification serves as the starting point from which to *derive* the program. This is the objective of much of the wide body of ongoing program transformation research.

Research on formal specification has tended to focus on two aspects, *using* specifications (the categories outlined above), and *expressing* them. Since the specifications of complex tasks may themselves be complex (e.g., see the diverse case studies in [10]), expressing and comprehending such specifications may be difficult (in spite of the fact that specifications need state only what the task is, not how it is to be done). Specification languages, by providing a rich vocabulary, mitigate this difficulty, but do not eliminate it entirely. (We found this to be the case for our own specification language, Gist, even though it had been tailored from the outset to meet our abstract design criteria for an ideal specification language; our criteria are outlined in [3], and Gist is described in its early stages in [12], and in its more complete form in [4].) The inescapable conclusion is that the *process* of specification construction must be formalized and supported by mechanical tools; this is the area addressed by the remainder of the paper.

B. Supporting the Construction of Specifications

Early work toward facilitating construction of specifications was done by Burstall and Goguen; they argued that complex specifications should be put together from simple ones, and developed their language CLEAR to provide a mathematical foundation for this construction process [5]. They recognized that the construction process itself has structure, employs a number of repeatedly used operations, and is worthy of explicit formalization and support.

Goldman observed that natural language descriptions of complex tasks often incorporate an evolutionary vein—the final description can be viewed as an elaboration of some simpler description, itself the elaboration of yet a simpler description, etc., back to some description deemed sufficiently simple to be comprehended from a nonevolutionary description [11]. He identified three "dimensions" of changes between successive descriptions: *structural*—concerning the amount of detail the specification reveals about each individual state of the process, *temporal*—concerning the amount of change between successive states revealed by the specification, and *coverage*—concerning the range of possible behaviors permitted by a specification.

Balzer has provided a complete characterization of generic changes to the structure of a domain model, and has begun to consider propagating the effects of those changes through the operations that use the model and through the already-established data base of information [2]. These changes are applied as high-level editing steps in the maintenance of the computing environment that we are using on a day-to-day basis.

Recently Fickas has begun to extend his AI problem-solving approach, originally for transformational derivation of implementations from specifications [8], to specification construction [9]. He identifies some domain-independent goals of specification construction, and methods to achieve them. Fundamental to his approach is the notion that the steps of the construction process can be viewed as the primitive operations of a more general problem-solving process, and are hence ultimately mechanizable.

Also somewhat related is the ongoing Programmer's Apprentice project (see [15] and, more recently, [19]). Although the approach to software production embodied in the aims of this project (to build a tool which will act as an intelligent assistant to a skilled programmer) differs from a specification-centered methodology, much of what they have found has relevance to our enterprise.

Finally, we see much overlap with issues arising in transformational derivations of implementations from specifications. Although such derivations are intended to preserve meaning across steps (whereas the elaboration steps in developing specifications deliberately *change* meaning), researchers have suggested the need to capture more of the structure of the development process—see [20] for Wile's exposition of these themes.

C. Incremental Elaboration of Specifications

We advocate that formal specifications be constructed and explained in the evolutionary style identified by Goldman, that is, start with a specification of some very simple, idealized version of the task, and incrementally elaborate that toward the final, complex specification. This has a number of potential benefits, discussed next.

Gradual Path to the Final Specification: Construction and comprehension can be done gradually. We do not have to introduce or digest all details of all aspects of the specification in one fell swoop, but rather can evolve our understanding from a simple starting point, at each stage introducing only a palatable amount of additional information.

Elaborations for Refinements and Adaptations: Some elaboration steps will be strightforward "refinements," simply adding more detail to the already denoted behaviors. Other steps will change which behaviors are denoted, by discarding some existing ones or adding new ones (which are not refinements of existing behaviors). For example, specification of a communication system may begin with an idealized version in which the communication channel is perfect, and an elaboration may retract this idealized assumption by adding in new behaviors in which transmissions become corrupted or fail entirely. The ability to make such adaptive changes liberates us from the constraint that our starting point be a pure abstraction of the final specification.

Idiomatic Elaborations: While the individual details affected by an elaboration depend upon the particulars of the task being specified, it is often the case that an elaboration causes some change whose high-level nature can be expressed somewhat independently of the task. For example, an elaboration might add a level of subtypes to an existing data type, or introduce an exception to the already described normal-case behavior. At this more abstract level of description, the same changes may occur repeatedly. Recognizing and recording the structure of the elaboration process in terms of these abstractions should facilitate both explanation and maintenance of a specification.

High-Level Editing, Mechanized Support for Elaborations: Further study of idiomatic elaborations reveals that the intended change can often be described fairly concisely, while the execution of that change involves the repetition of individual changes distributed across the specification. This phenomenon suggests an opportunity for mechanized tools to support performing the change. In this respect our approach resembles that of the Programmer's Apprentice project, wherein the Knowledge-Based Editor in Emacs (KBEmacs) automatically takes care of several kinds of programming details involved in a change.

Enhanced Potential for Reuse: We foresee an expanded opportunity for *reuse* of specification components. Abstract, general-purpose components can be introduced and thereafter adjusted and tailored to the specifics of the particular task. Note that elaborations that *adapt* a specification's behavior more than simply refining it will be used to do the adjustment and tailoring of components. Similarly, in the Programmer's Apprentice project, programming "cliches" are combined to construct programs, and introduction of a cliche is followed by editing to tailor it to the particular programming task, just as (we are arguing) it is necessary to tailor introduced specification components.

Easier Maintenance: Maintenance (evolution) of specifications can be achieved by adding, removing, or modifying the elaborations, and thereafter rederiving the specification. We believe that the structured nature of the elaboration process and the concise statements of elaborations will aid in locating and performing modifications. Of course, modifications to early elaborations may have far-reaching consequences if successive elaborations rely heavily upon some aspect that has been adjusted. The elaboration approach (or any other) cannot make specifications infinitely transmutable at little cost.

D. Conducting Independent Elaborations in Parallel

A further aspect of our approach is a degree of arbitrariness in the ordering of *independent* elaborations, that is, elaborations that do not depend upon each other. In such a case, incremental construction (or explanation) can be divided into a separate path for each of the independent elaborations, and each path pursued in *parallel*. This forms a *tree* of elaboration paths, whose branch points correspond to the divergence of parallel elaborations, and whose leaves correspond to specifications, each elaborated in some manner particular to its branch. These specifications must somehow be combined to realize the complete, final specification.[1] We see a number of advantages to formalizing and supporting this style of parallel elaboration followed by combination, discussed next.

Separation of Concerns: Separately considering parallel elaborations of a specification reduces the number of details that must be considered at any one time by excluding extraneous details. Thus the anticipated advantages of the incremental approach should be amplified by this parallelism.

Explicit Combination: Combination of completely independent parallel elaboration paths should be trivial, and hence completely automatable. However, complete independence is a perfect ideal; in practice, elaborations may often be "almost" independent, in the sense that there is some small amount of interaction among them. In such cases, it *is* worthwhile to pretend that they are independent, delaying consideration of minor dependencies until it is time to combine them. Because combination is an explicit activity, it increases the likelihood that we will recognize and think about such dependencies, which manifest themselves as further choices. Mechanized tools might be able to suggest those choices, and assist in executing the one chosen by the specifier.

II. Case Study—The Patient Monitoring System

This section presents the elaboration-style development of the specification of a small system-modeling example, the Patient Monitoring System. The example has been used by a number of researchers (including Stevens, Myers, and Constantine [17]; Zave [21]; and Rotenstreich and Howden [16]) to illustrate approaches to specification and implementation. The original statement of the example is:

> A patient-monitoring program is required for a hospital. Each patient is monitored by an analog device which measures factors such as pulse, temperature, blood pressure, and skin resistance. The program reads these factors on a periodic basis (specified for each patient) and stores these factors in a database. For each patient, safe ranges for each factor are specified (e.g., patient X's valid temperature range is 98 to 99.5 degrees Farenheit). If a factor falls outside of a patient's safe range, or if an analog device fails, the nurse's station is notified. [17].

The specification of the Patient Monitoring System presented below follows our suggested approach, that is, starts from some trivial initial specification and is elaborated in a number of parallel ways which are thereafter combined to result in the final, complete specification.

A. Starting Point

The starting point is a very simple specification that describes an approximation to the actual task. It is intended to be simple enough to be easy to comprehend. Initially, it comprises the following:[2]

- *Patients*, of which there may be several. Each has a "value" (a crude approximation of that patient's medical status) which changes randomly.
- *The nurse's station*, of which there is only one. It contains the definition of whether or not a patient's value is considered safe.[3] It receives notifications from the monitor of unsafe patient values.
- *The monitor*, of which there is only one. It watches the value of each patient, and upon the value becoming "unsafe" (whether a value is safe or unsafe is determined by the definition belonging to the nurse's station), notifies the nurse's station. The monitor is the piece to be implemented.

Fig. 1. shows the starting point expressed in our specification language Gist.[4] As we perform the elaborations, the changes to this Gist specification will be presented.

Closed System Specification: The piece to be implemented (the monitor) is described along with some details of its environment (patients and the nurse's station). By locating the randomly changing patient values and the def-

[1]When a later elaboration refers to details that are introduced in several earlier, parallel elaborations, their paths have to be combined in preparation for applying the later elaboration. Thus more generally the structure of the elaboration process is a *lattice* rather than simply a tree.

[2]An even simpler starting specification would have had just one patient, leaving it as an elaboration step to extend the specification to multiple patients. However, the starting point presented is sufficiently simple, and provides enough examples of elaborations to be instructive yet not overwhelming.

[3]The English description merely states that ". . .safe ranges. . .are specified. . ."; I have taken this to mean that their specification is part of the monitor's environment; the nurse's station seems the appropriate part of the environment to contain their specification.

[4]Occasionally Gist's syntax has been simplified, and some of the details of the specification have been omitted (notably declarations of cardinality of relations and types—e.g., that there is one and only one nurse's station, and of the interfaces between agents—e.g., that the monitor may observe but not change the definition of SAFE that resides in the nurse's station).

```
{ type value
      A new type called value is declared (Gist language keywords are underlined throughout and
    these italics are used here to comment the Gist specification).

.
 type patient with
 { implicit relation PVALUE(patient,value) }
    Patient is declared a type, along with a subsidiary declaration of a relation named PVALUE.
    PVALUE is a binary relation between objects of type patient and objects of type value. The
    keyword implicit indicates that tuples of the relation are inserted and deleted
    nondeterministically (modeling the unpredictable variability of patients' status).

.
 type monitor with
 { demon NOTICE_UNSAFE[patient]
     when start not SAFE( PVALUE(patient,?) )
     do NOTIFY[patient]
 }
      Monitor is declared a type with a subsidiary demon NOTICE_UNSAFE, parameterized
    by type patient. A demon has a predicate (following the keyword when) and a statement
    (following the keyword do). In every state, for each possible instantiation of its parameter(s)
    that makes its predicate true, a demon begins its statement. "PVALUE(patient,?)" denotes
    any object o for which PVALUE(patient,o) holds (in this case, a uniquely determined object
    of type value). "start P" where P is a predicate is true if P has just become true. Thus the
    NOTICE_UNSAFE demon will invoke the NOTIFY procedure on a patient in every state in
    which that patient's PVALUE has stopped being safe.

.
 type nurse's_station with
 { relation SAFE(value) , procedure NOTIFY[patient] }
      Nurse's_station is declared a type, with a subsidiary relation SAFE on values; NOTIFY
    is a subsidiary procedure which takes one argument of type patient and does nothing
    (however calls to the procedure are part of the denotation of the specification).
}
```

Fig. 1. Starting point specification.

inition of safe and unsafe values in the environment, it is clear that the monitor must respond to these, but cannot affect them (e.g., cannot redefine safe).

Together, the monitor and its environment form an executable model of the proposed system interacting with its environment. This model is *operational*, in that it defines ongoing behaviors in which patient values are changing and the monitor is responding appropriately. In fact, the specification models every possible behavior of the environment (changing patient values, and definitions of safety of values), and thus completely defines how the monitor must behave. We believe this to be a useful way of specifying so-called "embedded systems." In this respect, we strongly agree with Zave, who calls this an "operational" approach to specification; she uses the patient monitoring system and other examples as illustrations [21]. Our specification language, Gist, has been constructed to facilitate this approach to specification (see [7] for more discussion of this issue). Futher mention here will be limited to pertinent interactions with the elaborative style.

B. The Elaborations

The initial specification is a very crude approximation of the patient monitoring world. The elaborations that gradually develop the approximation are as follows:

- Splice in a device between each patient and the monitor, so that the monitor reads values of devices rather than patients, and each device forwards its patient's value.
- Introduce the possibility that devices fail (this elaboration depends upon the results of the previous one).
- Refine a patient's value into a composite of values, one for each factor.
- Modify the monitor from observing continuously to observing periodically.
- Make the safety of a patient's value depend on which patient it is.

These will be explained in detail in the sections that follow, and for each, the following issues will be addressed:

Motivation: The elaborations are justified in terms of the domain of monitoring patients.

Change: An elaboration may change the meaning of the specification, i.e., the behaviors it denotes. Goldman's three dimensions of change will be used as the first level of categorization of the changes encountered in this example.

High-Level Editing Step: Elaborations are shown to be instances of high-level editing steps, and we speculate on the role of automated support for executing these steps.

Continuation: Some further elaborations that might be necessary to accurately model the task domain are mentioned but not explored in depth within this paper.

1) Elaboration of Monitor's Reading of Patient Values—Splicing in a Device Between Patient and Monitor: The monitor may not read each patient's value directly, but must instead read the value of an intermediary, a "device" that continuously observes a patient's value, and displays that value (unchanged) to the monitor.

Motivation: This elaboration is made to refine the model of the monitoring domain, in this case concerning the monitor's observation of patients.

Change: At this stage, since a device continuously displays its patient's value, the monitor will react as before, indeed, the whole system will operate as before; the only change has been to introduce a level of indirection into some internal communication. As future elaborations approximate devices, this will no longer be the case. In Goldman's terms, it is a pure *structural* change because it increases the detail within the states of the behaviors, but does not change the number of states, nor the set of behaviors.

High-Level Editing Step: We expect the splicing of an intermediary into a previously direct data path to recur frequently in specification elaborations. We envisage describing the desired change to an automated tool which would then perform the change at all the requisite locations in the specification. The new and modified portions of the specification are shown in Fig. 2 (boldface is used to highlight the changes).

2) Elaboration of Devices—Devices May Fail: This elaboration refines devices (and so must follow the elaboration that introduced devices into the specification). A device was defined to display its patient's value. The elaboration introduces a new behavior for a device, which is to be thought of as an *exception* to the already defined *normal case* behavior. The new behavior is that a device may fail; while a device is failed, it displays *any* value, not necessarily its patient's value. This elaboration is presented in two stages: **introduction** of the exceptional behavior, and **adjustment** of the specification in response to the introduced exception.

Introduction of exceptional behavior (device failure):

Motivation: This refines the model of the environment to reflect the reality of imperfect devices.

```
type device with
{ implicit relation DVALUE(device,value)
,
  invariant DEVICE_VALUE_EQUALS_LINKED_PATIENT_VALUE
     all d|device || DVALUE(d,?) = PVALUE(LINK(d,?),?)
,
  relation LINK(device,patient)
}
```

Device is declared a type with subsidiary relation DVALUE between devices and values. Similarly, LINK is a relation between devices and patients. DEVICE_VALUE_EQUALS_LINKED_PATIENT_VALUE is an invariant, a predicate which must hold in every state. It is used here to ensure that for every device d, d's value (namely, the value in the DVALUE relation to d) equals d's patient's value (namely, the value in the PVALUE relation to the patient in the LINK relation to d).

```
,
type monitor with
{ demon NOTICE_UNSAFE[patient]
    when start not SAFE( DVALUE(LINK(?,patient),?) )
    do NOTIFY(patient)
}
```

Fig. 2. Modifications to introduce devices.

Change: Device failure adds more detail (each device is in one of two conditions, failed or not failed), and adds new behaviors that are not refinements of any of the original behaviors (while a device is failed, the value it displays need not be the same as its patient's value; as a consequence, the occasions on which the monitor issues warnings to the nurse's station may be different from before). In Goldman's terms, the change affects both the structural and coverage dimensions.

High-Level Editing Step: Introduction of the exceptional case is straightforward—a unary relation (i.e., a predicate) on devices models whether or not they are failed, entry into and out of failed state is specified to occur at random, and the invariant relating a device's value to its patient's value is weakened to hold only when the device is not failed (i.e., does not hold in the exceptional case). Use of a unary relation is likely to be a frequently applied characterization of an exceptional condition; if so, an automated editing step could assist in introducing such a relation, and in applying it to weakening an invariant in this manner. The modified portions of the specification are shown in Fig. 3.

Adjustment of the specification in response to the introduced exceptional behavior:

To make the adjustment, we determine where the change to devices impacts the specification, adjust those locations as appropriate, and continue by propagating the impact of those adjustments recursively.

Motivation: One of the underlying goals of the specification is that the nurse's station is accurately informed of the status of patients. The failure of devices violates that goal, and thus prompts us to consider adjusting the specification accordingly.

High-Level Editing Step: Determining where the change impacts the specification could be done automatically by locating those places where devices are used, namely within the declaration of the LINK relation, and within the NOTICE_UNSAFE demon of the monitor. We decide that declaration of the LINK relation needs no adjustment (a device remains linked to the same patient irrespective of whether or not the device is failed). Within the NOTICE_UNSAFE demon, we decide to adjust the predicate of the conditional that determines when NOTIFY is invoked. For the normal case (not failed), we leave the condition unchanged; for the exceptional case (failed), we cause NOTIFY to be invoked at the start of that exception; this is shown in Fig. 4.

```
type device with
{ implicit relation DVALUE(device,value)
,
  implicit relation FAILED(device)
,
  invariant DEVICE_VALUE_EQUALS_LINKED_PATIENT_VALUE
     all d|device || not FAILED(d) implies
                     DVALUE(d,?) = PVALUE(LINK(d,?),?)
,
  relation LINK(device,patient)
}
```

Fig. 3. Modifications to introduce device failure.

In adjusting an impacted portion of the specification there may be a number of reasonable alternatives for an automated tool to suggest, among which the specifier would select.[5]

Change: Our choice of adjustments seems a reasonable compromise of the original goal, since in the normal case (device not failed) things happen as before, and in the exceptional case (device failed) a notification is issued as soon as the exceptional case arises.

Continuation: Because NOTIFY now can be invoked in two cases, in the normal case of an unsafe patient value arising, and in the exceptional case of device failure, a reasonable extension would be to distinguish between these cases when NOTIFY is invoked. This could be done by extending NOTIFY with an additional boolean argument to indicate whether it had been invoked in the exceptional case or not; see Fig. 5.

Device failure is assumed to be a detectable condition, detectable in some manner other than by comparing its displayed value with its patient's value (the monitor, not having access to the patient values, cannot do such a comparison; instead, detection of such failure is represented by giving the monitor access to the unary relation FAILED). In reality, a device might fail without giving any such indication, in which case the monitor could do nothing to detect such an occurrence, and would not know to warn the nurse's station on an unsafe condition arising. This is easily modeled by, say, introducing (via an elaboration, of course) a secondary unary relation on devices that acted like FAILED but could not be seen or used by the monitor. This would extend the behaviors denoted by the specification to include some in which a device no longer reflects its patient's value, yet its FAILED relation remains false, in which circumstances the patient value may be unsafe without the nurse's station ever having been notified. Such a specification would draw our attention to the possibility that perfectly safe monitoring may be impossible given devices that fail undetectably, but would not alter the way the monitoring program should respond to the signals it is able to observe.

3) Elaboration of Patients—A Composite of Values, One for Each "Factor": We now turn our attention to

[5]One such alternative in our example would be to provide relation SAFE with both the device value and an indication of whether or not the device was failed, and leave it to the definition of SAFE, i.e., the nurse's station, to decide whether or not these constituted a safe condition.

```
demon NOTICE_UNSAFE[patient]
when not FAILED(LINK(?,patient))
                              and start not SAFE( DVALUE(LINK(?,patient),?) )
     or start FAILED(LINK(?,patient))
do NOTIFY[patient]
```

Fig. 4. Modifications to adjust NOTICE_UNSAFE to respond to device failure.

```
demon NOTICE_UNSAFE[patient]
when not FAILED(LINK(?,patient))
                                    and start not SAFE( DVALUE(LINK(?,patient),?) )
     or start FAILED(LINK(?,patient))
do NOTIFY[patient , FAILED(LINK(?,patient)) ]
...
NOTIFY | procedure[patient,boolean]
```

Fig. 5. Modifications to distinguish normal and exceptional NOTIFY invocations.

the patients themselves, and elaborate the "value" that approximates the medical status of a patient. Since this elaboration is independent of any devices (or lack thereof) acting as intermediaries between patients and the monitoring system, it can be explored in *parallel* with the device elaboration. See Fig. 6 for the growing structure of elaborations.

Motivation: A patient has not just one value, but several—one for each "factor" (pulse, temperature, blood pressure and skin resistance). The safety of a patient is the conjunction of the safety of that patient's individual factor values.

High-Level Editing Step: We expect refinement of a single value into a composite of values to be another frequently recurring step in specification elaboration. A high-level editing mechanism to perform this elaboration would assist the specifier in modifying the declaration of the value, introducing the new declaration of the composite, and modifying the uses of that value throughout the specification. Some input would be required from the specifier, namely indication of the value to be redefined, delineation of the range of components of the new composite, and direction of whether to replace uses of the single value with the composite, or with a (further selected) combination of the individual values of the composite.

Fig. 7 shows the new declarations (types "composite" and "factor," where factor is defined to be the range of named factors that are to be monitored, and relation "CFV" to relate a composite object to factors and their corresponding values), the adjusted PVALUE relation of patients (adjusted to relate a patient to a composite instead of simply a value), and the adjusted use of PVALUE within the NOTICE_UNSAFE demon.

We have chosen to leave SAFE dealing with values, and within NOTICE_UNSAFE combine the safety of all the factor values to determine the safety of the composite.[6] We have further chosen a combination that makes

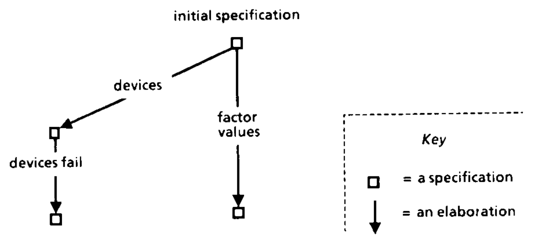

Fig. 6. A parallel elaboration—refining patient value into factor values.

```
type factor = { 'pulse, 'temperature, 'blood_pressure, 'skin_resistance }
,
type composite
,
relation CFV(composite,factor,value)
...
implicit relation PVALUE(patient,composite)
...
demon NOTICE_UNSAFE[patient]
  when start not all f | factor || SAFE( CFV(PVALUE(patient,?), f, ?) )
  do ...
```

Fig. 7. Modifications to introduce composite of factor values.

safety of a composite of patient values be the conjunction of the safety of its individual factor values.[7] This induces the universal quantification over factors.

We are making the assumption that the safety of each factor value is determined by the value alone; more likely, safety is a function of both the value and the identity of which factor it is, so we extend SAFE with an extra argument, the factor:

```
demon NOTICE_UNSAFE[patient]
  when start not all f | factor ||
               SAFE( CFV(PVALUE(patient,?),f,?), f )
  do . . .
. . .
relation SAFE(value,factor)
```

Change: This elaboration changes the meaning of the specification by adding more detail to the contents of states—in Goldman's terms for describing the evolution of specifications, it increases the *structural* granularity.

Continuation: We could continue elaboration of values by asserting that the range of values for a factor is linearly ordered, and that a "safe" value is any value between an upper and lower bound of values. Then the safety table maintained by the nurse's station need specify only these upper and lower bounds, leaving it to the monitor to compare the value it has with the bounds. This elaboration leads to the situation described in the English statement of the monitoring system; however, for the sake of simplicity in this presentation, it is omitted.

4) Elaboration of the Monitor—Reads Periodically Rather Than Continuously: At this point we turn our attention to the monitor, and consider its task of reacting to a number of changing values. The elaboration is to have

[6]An alternative is to pass the composite to SAFE, leaving the nurse's station to determine the safety of a patient given the composite of that patient's values:

```
demon NOTICE_UNSAFE[patient]
  when start not SAFE( PVALUE(patient,?) )
  do . . .
. . .
relation SAFE(composite)
```

[7]Other obvious means for combining properties of components include: disjunction (e.g., a composite is unsafe if any of its values is unsafe), majority (e.g., the economy is in a slump if more than half its indicators are in a slump), the average (which would involve computing an average value from the multiple component values, and then inspecting whether the property held of that average).

the monitor take observations at discrete intervals rather than continuously. Since this elaboration is independent of whether the monitor is observing patient values or device values, and independent of whether a patient has one or several values, it can be done in *parallel* with the device elaborations and the patient value elaboration. See Fig. 8 for the growing structure of elaborations.

Motivation: The motivation for this elaboration arises from considering the realities of implementing the monitor software that is supposed to react instantaneously to changing values, sending off notifications when appropriate. An implementation might not be capable of reacting instantaneously, particularly if it is responsible for monitoring a number of patients. Hence, motivated by implementation concerns, we should modify the specification. We may decide that the monitor should read each patient's value at an appropriately chosen time period (generous enough to permit the monitor to schedule the activities it has to perform), where the period for a patient is set by the nurse's station.

This is an instance of the *intertwining* between specification and implementation, a frequent phenomenon, according to Swartout and Balzer [18], and hence one that we must be able to accommodate within our formalization and mechanization of the software development process.

Change: This elaboration step also alters the overall meaning of the specification, by introducing the possibility that a patient's value be unsafe without the nurse's station having been warned (possible if the monitor has not yet reread the patient's value since it changed). In Goldman's terms, this elaboration changes the *coverage* (the range of possible behaviors).

Observe that this change to coverage is an *implied consequence* of the elaboration, and is not explicit in the formulation of the elaboration (in contrast, the elaboration that introduced the possibility of device failure explicitly added the new, exceptional, behaviors). Hence it is crucial to *recognize* this implicit change. Two factors combine to increase the likelihood of detection—the closed system style specification, in which the environment (of patients, devices, and nurse's station) is at least partially described, thus formalizing enough of the entire system to capture the aspects that have changed, and the style of parallel elaborations, in which each elaboration is a relatively small increment to a specification containing only relevant details, thus increasing the likelihood that we can discern all the ramifications of the elaboration.

High-Level Editing Step: This change makes a continuous process, observation of some value, discrete; presumably, modifying any continuous observation in this fashion is also useful to offer as a mechanized option.

The changes to our example are shown in Fig. 9. Agent "clock" has been added to simulate the passage of time, and the monitor has been augmented with a demon to read a patient's value into the LATEST_VALUE_READ relation at periodic intervals (relative to the start time for reading that patient's value). Both the start time and the period are determined by the nurse's station. Reference to

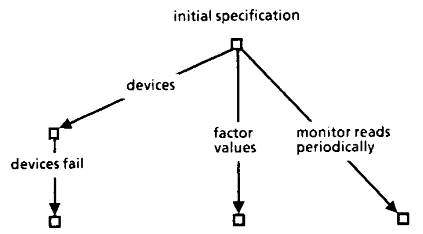

Fig. 8. Another parallel elaboration—the monitor reads periodically.

```
type clock with
{ relationCLOCK_TIME(integer)
,
  demonTICK_TOCK[]
    when true
    do choose { CLOCK_TIME(?) := CLOCK_TIME(?) + 1, null }
}
     Clock is an agent with one relation, CLOCK_TIME, whose value is occasionally
    incremented (in every state demon TICK_TOCK nondeterministically chooses to increment
    the value or do nothing).
,
type monitor with
{ relation LATEST_VALUE_READ(patient,value)
,
  demon READ_PERIODICALLY[patient]
    when CLOCK_TIME(?) = START_READING_TIME(patient,?) +
                         (any integer) * PERIOD(patient,?)
    do LATEST_VALUE_READ(patient,?) := PVALUE(patient,?)
,
  demon NOTICE_UNSAFE[patient]
    when start not SAFE( LATEST_VALUE_READ(patient,?) )
    do NOTIFY[patient]
}
,
type nurse's_station with
{ relation SAFE(value) , procedure NOTIFY[patient]
,
  relation START_READING_TIME(patient,integer)
,
  relation PERIOD(patient,integer)
}
```

Fig. 9. Modifications to change the monitor to read periodically.

a patient value has been replaced by a reference to the latest value read for that patient.

The clock and the READ_PERIODICALLY demon (and its associated relations) might exist as reusable components that are brought in and tailored to the specification; a high-level editing step could be used to adjust references to the patient's value to refer instead to the latest read value.

5) Elaboration of the Safety of a Patient's Value—Depends on Which Patient It Is: The definition of patient safety is refined to be dependent on the patient. Again, this is parallel to all the other elaborations; their structure is shown in Fig. 10.

High-Level Editing Step: Parameterizing a construct and its uses to be dependent on an extra argument is a simple high-level editing step. It necessitates extending the declaration of the construct, and extending uses of the construct throughout the specification by providing the extra argument (usually an obvious suggestion for the value can be determined from the context).

Here, the relation modelling patient safety, SAFE, is parameterized throughout the specification with the identity of the patient. Where SAFE is used (in the NOTICE_UNSAFE demon), the context provides the patient whose safety is being checked. The changes are shown in Fig. 11.

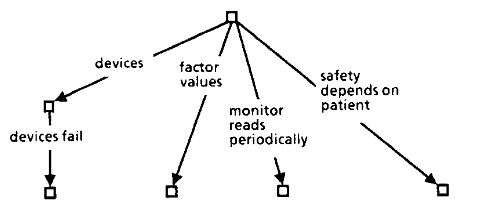

Fig. 10. Another parallel elaboration—safe depends on which patient it is.

```
type monitor with
{ demon NOTICE_UNSAFE[patient]
    when start not SAFE( PVALUE(patient,?) , patient )
    do ...
}
...
type nurse's_station with
{ relation SAFE(value,patient) ... }
```

Fig. 11. Modifications to elaborate safe to depend on which patient it is.

Change: This simple step adds more detail to the contents of states, i.e., is one of Goldman's *structural* changes.

C. Combining the Elaborations

Development of the specification from this point continues by combining the parallel elaborations described so far.[8] The goal of combination is a single specification that merges the effects of all the divergent elaboration paths.

1) Achieving Combination: Combination of parallel elaborations is achieved by sequentially applying all the steps of those elaborations. Where the parallel elaboration structure implies some ordering, sequential application of the steps must, of course, abide by the ordering. For example, in our parallel elaboration of the patient monitoring system, introduction of devices preceded device failure, hence in sequential application device introduction must be done before device failure (but may be interleaved with steps from other elaboration paths).

Whatever guidance was input by the specifier during exploration of the parallel elaborations can be automatically replayed to direct their reapplication in this sequential process.

2) Options During Combination: If the parallel elaborations were completely independent, combination could be done entirely automatically. When dependencies do arise, these manifest themselves as further options among which the specifier must select (e.g., combining the elaborations of introducing devices and of refining the single patient value to multiple factor values offers options of having a single device per value, or a single device reading the whole composite of values).

We believe that the specifier can recognize dependencies in advance of actual combination by comparing pairs of elaborations drawn from parallel paths. On recognizing a dependency, the specifier then selects from among the options offered by that dependency. When elaborations have a high-level characterization, it is often the case that their combination also has a high-level characterization, with which there may be associated a recurring range of options. As with high-level editing steps, we postulate the beneficial role of automated mechanisms, in this case to help recognize dependencies and identify the range of options.

Having detected all the dependencies and made all the choices, combination by the sequential application of elaborations can proceed without the need for further input from the specifier.

3) Combination of the Patient Monitoring System Elaborations: The structure of all the parallel elaborations to be combined was shown in Fig. 10; Fig. 12 presents a table showing pairwise comparisons of these elaborations. Note that only those pairs drawn from different elaboration paths are compared, thus, for example, introducing devices and device failure need not be compared. The comparisons that reveal dependencies are discussed in detail next.

Comparing Elaboration of Devices and Factor Values: This comparison is between the elaboration that introduced devices with the elaboration that refined a patient's value into a composite of values, one for each factor. We recognize options of having a patient's device read that patient's composite of values, or having several devices linked to a patient, one to read each factor value (or something in between, with several devices, each reading several factor values). The appropriate choice must be provided by the specifier, depending as it does upon the real-world domain of devices. If the former, the devices are trivially changed to deal consistently with composites rather than individual values. If the latter, each device is parameterized[9] with the factor whose value it reads.

High-Level Characterization: This is a comparison of two elaborations, one of which refines a value into a composite, the other of which splices an intermediary into a data path transmitting the objects being refined. Clearly, there is a choice between having the intermediary transmit the composite resulting from the refinement, or splitting the intermediary into several intermediaries, one to transmit each component of the refined type.

Comparing Elaborations of Factor Values and Monitor Observing Periodically: We recognize options of having the monitor read different factor values at different times (by staggering the starting times for reading factor values) and/or at different periods. This is similar to the previous comparison, in that the value refinement may be used to parameterize details introduced by the other refinement (in the previous comparison, devices; here, the periodicity of observations).

[8]A further aspect of the English statement of the problem description, storing the readings in a data base, will be introduced later to illustrate specification maintenance, Section II-D.

[9]The same high-level editing step as was used to parameterize SAFE with the identity of the patient, Section II-B-5.

safety depends on patient					
periodic monitoring	*independent*				
factor values	*independent*	*period depends on factor?*			
devices fail	*independent*	*periodic monitoring of failure?*	*independent*		
devices	*independent*	*independent*	*different device for each factor?*	*(already considered sequentially)*	
	safety depends on patient	periodic monitoring	factor values	devices fail	devices

Fig. 12. Comparison of parallel elaborations to determine options.

High-Level Characterization: Comparison here is between a value refinement and a change of a process from continuous to periodic. If the process involves the refined value, this suggests the option of parameterizing the newly introduced aspects of the discrete process with that refinement.

Comparing Elaborations of Device Failure and Monitor Observing Periodically: The options revealed by this comparison are to have the monitor watch for device failure continuously or periodically. If the latter is chosen, two further options arise: whether the monitor should observe devices for failure at the same time as it reads device value(s), or to define different starting times and/or periods for this other activity.

High-Level Characterization: This is a comparison of exception introduction with modification of a process from continuous to periodic, where the exception impacts this process. It offers the option of modifying the processing of the exception to also be periodic.

D. Maintenance

There can be a number of reasons for later wanting to *change* the specification: feedback from the attempt to derive an efficient implementation, revision of requirements based on experience with the operating program, or simply some domain change (e.g., a new class of devices that behave in some different manner). A major tenet of the transformation-based approach to software development is that the change to the implementation is achieved by changing the specification and rederiving the program, not by changing the program directly (see [1] and [6] for discussion). This presupposes that we are able to make the appropriate change to the specification. Incorporation of changes fits in very well with the elaboration-based approach, since a change can be introduced as an adjustment to one of the steps, or added as a new step at the appropriate location within the development structure. In either case some or all of the combination process may need to be reconsidered or extended.

For purposes of illustration, the reference to storing factors in the original statement of the task—"The program reads these factors on a periodic basis (specified for each patient) *and stores these factors in a database*"—has been omitted in the preceding development, and is introduced now as if it were an unanticipated addition to the specification. Adhering to our style, we incorporate this as an addition to the tree of parallel elaborations and thereafter consider its impact on the combination process.

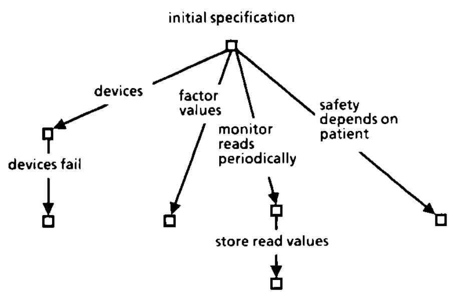

Fig. 13. Another elaboration—store read values.

```
type monitor with
{ relation LATEST_VALUE_READ(patient,value)
,
  demon READ_PERIODICALLY[patient]
   when CLOCK_TIME(?) = ...
   do LATEST_VALUE_READ(patient,?) := PVALUE(patient,?) ;
      STORE[patient, LATEST_VALUE_READ(patient,?)]
,
  demon NOTICE_UNSAFE[patient] ...
}
...
type nurse's_station with
{ ... , procedure STORE[patient,value] }
```

Fig. 14. Modifications to store read values.

We seek to add it to a point as close as possible to the root of the tree of elaborations, that is, as independent of the other elaborations as possible. It is important to maximize such independence because it permits us to take full advantage of our high-level editing tools in performing this addition and any future maintenance. We place this addition following the elaboration of the monitor from continuous to periodic, since storage is coupled to reading of values. The extended structure of parallel elaborations is shown in Fig. 13, and the modified portions of the specification in Fig. 14, in which the read demon has been extended to cause it to invoke STORE (a new procedure of the nurse's station), passing as parameters the patient and the value read.

We incrementally effect the combination of this additional step with the other parallel elaborations by comparing it with them to recognize any dependencies, selecting among any options that dependencies offer, and applying the selected version. Comparison is shown in Fig. 15. The only dependency we recognize is combining the storage of values and device failure: should we store values read from failed devices and/or store a record of device failure? The other combinations are independent, and can be effected automatically (e.g., combining storage of read values and introduction of devices leads to storage of the values read from the devices rather than from patients).

store read values	*independent*	*(already considered sequentially)*	*independent*	*also store record of failure?*	*independent*
	safety depends on patient	periodic monitoring	factor values	devices fail	devices

Fig. 15. Comparing store read values elaboration with other elaborations.

III. Discussion

We now review the overall methodology, summarize the findings from our hand-conducted case studies, and consider the open questions that must be answered to establish the viability of this whole approach.

A. Summary of Methodology

We have suggested that specifications be constructed and explained incrementally, by applying elaborations to an initial specification that is simple enough to construct or explain at once. The elaborations effect refinements and adjustments to the evolving specification. Often they have some high-level characterization. It is intended that a mechanized system be employed to assist in applying the elaborations. When elaborations are independent, it is recommended that they be applied in parallel, giving rise to a tree structure rather than simply a linear structure of evolution. Multiple specifications emerge as the leaves of this tree, and must be combined to realize the final specification. This can be achieved by sequentially applying all the elaborations in any order consistent with the tree. Prior to this it may be necessary to compare the parallel elaborations to detect and resolve minor dependencies that offer multiple choices during combination.

B. Findings

We have done some preliminary studies in the form of hand-performed developments of several small-scale examples (the patient-monitoring system, a small inventory control program, and a coordinator of robot arms). These are encouraging, insofar as they tend to confirm some of our expectations:

Gradual Path to Final Specification: The developments readily break down into incremental elaborations, and independence among elaborations occurs. We have *not* attempted to empirically verify our claim that it is easier to construct or explain specifications by this incremental approach.

Recurring Idioms of Elaboration: The high-level nature of a number of elaborations is evident, and is repeated across specification developments, for example, refining a data type into a composite of values (a patient's single value into a value for each factor), and adding an exceptional case to normal behavior (device failure).

High-Level Editing: Our hand-performed studies, even of small-scale examples, clearly indicate that mechanized support is *essential!* The mundane, repetitious bookkeeping details of developing a specification by elaboration and combination quickly become intolerable when done by hand. Without underlying mechanization, this approach is not viable as a paper-and-pencil discipline. Fortunately, we can identify numerous plausible possibilities for mechanized support of the elaboration and combination steps. Our intent is to use Wile's POPART system as the framework in which to build these mechanisms. POPART provides facilities for manipulating grammatical objects (in this case, Gist specifications) and for performing, and recording the structure of, those manipulations [20]. In the past we have used this for experimenting with the transformational development of Gist specifications into programs.

Reuse: Few convincing examples of reuse of specification components have occurred, perhaps because of the paucity of case studies we have yet performed. Most of the reuse has been concentrated within the high-level editing steps.

Maintenance: The simple addition of another parallel elaboration was easy to incorporate. We also believe that the continuations (suggested as further elaborations to some of the steps) would also be easy to incorporate incrementally. We have yet to investigate more intricate modifications (e.g., a modification that *retracts* some effects of an earlier elaboration, such as deciding that devices for reading certain of the factor values do not fail); however, our intuition is that the flexibility provided by the incremental construction process will be beneficial to such maintenance.

C. Viability of Approach and Open Research Areas

If the elaboration approach is to be viable, the extra benefits it provides must compensate for the extra effort that it entails.

The formal and lengthy nature of the elaboration process is a potential source of considerable additional overhead (as compared to constructing specifications directly). Hence one of the primary goals is to introduce automation into this activity, which, if successful, will greatly reduce the overhead. The major role for automation is in applying the high-level editing commands—these should be mechanized to the point where little user input is required to guide their application once they have been selected. We anticipate that a number of high-level editing commands can be automated to this extent. Our intent is to build a library of such mechanized commands. Two crucial questions to be asked of such a library are:

Coverage: How successfully can the library provide coverage, that is, when faced with a new task to specify, will sufficiently many of the evolutionary changes already be present as high-level editing steps in the library? At present we do not even know what comprises "sufficiently many."

Indexing: Depending upon the size of the library, finding the right high-level editing command might be nontrivial. We need a taxonomy of high-level editing commands to serve as an index.

We expect that automation will play a lesser role in aiding detection of dependencies among elaborations (which is done prior to combining them), because this would seem to be a harder overall problem, but fortunately less of a source of additional effort, since no matter how a specification is constructed, it will be necessary to consider the potential dependencies among the portions of the specification. In this approach, the incremental nature of the process may force the examination of a larger number of cases for dependencies, but each case involves only the explicitly delineated effects of the elaborations being combined.

It may be the case that to reduce the cost of elaboration-style construction below that of direct construction requires a level of sophistication in the mechanisms that is unobtainable in the foreseeable future. However, if we can benefit from the record of evolution throughout the lifetime of the specification, that amortized benefit may outweigh the increased initial cost. One immediate benefit is the use of the recorded evolution for *explaining* the final specification. Another is the use of evolution as the basis for *maintaining* the specification. Since evolution records the specification's design, we may hope that in modifying the specification we can reuse the portions of the design that are unaffected by the modification, and adjust those that are affected. It should be easier to do this than to attempt to modify the specification without access to its design history. Replay of designs in the face of change is a major open research area, and it remains to be seen how far we can progress in this direction. See [14] for a discussion of this general issue.

One other open research area is how to combine the elaboration approach with other means for supporting specification construction, such as generalizing from examples, construction by analogy (when the task is similar to some previously specified task), and amalgamating specifications of the different viewpoints of a system to emerge with a specification of the whole system.

Acknowledgment

The research described here has been conducted in the framework of our software specification group, headed by B. Balzer and including D. Cohen, N. Goldman, E. Ipser, L. Johnson, B. Swartout, D. Wile, and K. Yue. They share the credit for generation and clarification of the ideas presented herein. The author has also benefited from feedback from IFIP Working Group 2.1, from discussions with S. Fickas and his research group at the University of Oregon, Eugene, and from constructive comments from J. Mostow. Thanks are also due to S. Coyazo for careful scrutiny of earlier drafts.

References

[1] R. Balzer, "Evolution as a new basis for reusability," in *Proc. ITT Workshop Reusability in Programming*, Newport, RI, 1983, pp. 80-82.

[2] —, "Automated enhancement of knowledge representations," in *Proc. 9th Int. Joint Conf. Artificial Intelligence*, A. Joshi, Ed., Los Angeles, CA, Aug. 1985, pp. 203-207.

[3] R. Balzer and N. Goldman, "Principles of good software specification and their implications for specification languages," in *Specification of Reliable Software*. Washington, DC: IEEE Computer Society, 1979, pp. 58-67.

[4] R. Balzer, N. Goldman, and D. Wile, "Operational specification as the basis for rapid prototyping," *ACM Software Eng. Notes*, vol. 7, no. 5, pp. 3-16, Dec. 1982.

[5] R. M. Burstall and J. Goguen, "Putting theories together to make specifications," in *Proc. Fifth Int. Conf. Artificial Intelligence*, Aug. 1977, pp. 1045-1058.

[6] J. Darlington and M. S. Feather, "A transformational approach to program modification," Dept. Comput. and Contr., Imperial College, London, Tech. Rep. 80/3, 1980.

[7] M. S. Feather, "Language support for the specification and development of composite systems," *ACM Trans. Programm. Lang. Syst.*, vol. 9, no. 2, pp. 198-234, Apr. 1987.

[8] S. Fickas, "Automating the transformational development of software," *IEEE Trans. Software Eng.*, vol. SE-11, no. 11, pp. 1268-1277, Nov. 1985.

[9] —, "A knowledge-based approach to specification acquisition and construction," Dep. Comput. Sci., Univ. Oregon, Eugene, Tech. Rep. 86-1, 1986.

[10] N. Gehani and A. D. McGettrick, Eds., *Software Specification Techniques*. Reading, MA: Wesley, 1986.

[11] N. M. Goldman, "Three dimensions of design development," in *Proc. 3rd Nat. Conf. Artificial Intelligence*, Washington, DC, Aug. 1983, pp. 130-133.

[12] N. Goldman and D. Wile, "A relational data base foundation for process specification," in *Entity-Relationship Approach to Systems Analysis and Design*, P. P. Chen, Ed. Amsterdam, The Netherlands: North-Holland, 1980, pp. 413-432.

[13] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich, "Report on a knowledge based software assistant," Rome Air Development Center, Tech. Rep. RADC-TR-83-195, Aug. 1983.

[14] D. J. Mostow, "Some requirements for effective replay of derivations," in *Proc. 3rd Int. Machine Learning Workshop*, Skytop, PA, June 1985, pp. 129-132.

[15] C. Rich, H. E. Schrobe, and R. C. Waters, "An overview of the Programmer's Apprentice," in *Proc. 6th Int. Joint Conf. Artificial Intelligence*, 1979, pp. 827-828.

[16] S. Rotenstreich and W. E. Howden, "Two-dimensional program design," *IEEE Trans. Software Eng.*, vol. SE-12, no. 3, pp. 377-384, 1986.

[17] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115-139, 1974.

[18] W. Swartout and R. Balzer, "On the inevitable intertwining of specification and implementation," *Commun. ACM*, vol. 25, no. 7, pp. 438-440, 1982.

[19] R. C. Waters, "The programmer's apprentice: A session with KBEmacs," *IEEE Trans. Software Eng.*, vol. SE-11, no. 11, pp. 1296-1320, Nov. 1985.

[20] D. S. Wile, "Program developments: Formal explanations of implementations," *Commun. ACM*, vol. 26, no. 11, pp. 902-911, Nov. 1983.

[21] P. Zave, "An operational approach to requirements specification for embedded systems," *IEEE Trans. Software Eng.*, vol. SE-8, no. 3, pp. 250-269, May 1982.

Martin S. Feather received the B.A. and M.A. degrees from Cambridge University, England, in 1975 and 1976, respectively, and the Ph.D. degree in artificial intelligence from Edinburgh University, Edinburgh, Scotland, in 1979.

In October 1979 he joined the Information Sciences Institute, University of Southern California, where he has since remained as a Research Scientist working on program specification and transformation. He is a member and the secretary of IFIP Working Group 2.1. His research interests center around formalizing and providing mechanized support for the programming process.